Захист <https://youtu.be/8El8fJT8DRg>

**ЛАБОРАТОРНА РОБОТА №2**

**ПОШУК ОПТИМАЛНОГО РІШЕННЯ ЗА ДОПОМОГОЮ ГЕНЕТИЧНОГО АЛГОРИТМУ**

**Мета.** - отримати навички розв’язання практичних задач за допомогою генетичних алгоритмів.

**2.1 Теоретичні відомості**

Генетичні алгоритми (ГА) (Holland, 1969-1990) спрощено моделюють процеси природної еволюції і засновані на стохастических принципах.

Генетичні алгоритми зводяться до виконання наступних етапів:

1. Ініціалізувати популяцію.

2. Обчислити значення критерію якості для кожної особини популяції.

3. Виконати процес відтворення для кожної особини популяції.

4. Виконати схрещування і мутацію для кожної особини популяції.

5. Перевірити умову завершення. Якщо її не виконано, то повернутися до п. 2,.

Реалізація ГА зводиться до операцій з рядками: копіювання рядків, заміни фрагментів рядків і інверсії бітів.

**Хід роботи**

1. Завдання зображено в таблиці 2.1

Таблиця 2.1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Варіант | Функція критерія якості | n | pc | pm |
| 14 | f(x)=-(3x2-12)2/256, x ∈ [0, 255] | 34 | 0.75 | 0.001 |

2. Для розрахунку тестової ітерації потрібно згенерувати масив з 34 масивами, що дорівнює кількості популяції. Кожна особа має 8 бітів, що задає інтервал від 0 до 255. Тож скористаємося генератором для формування масиву.

Далі нам потрібно розрахувати значення пристосування – значення функції. Тож, спочатку за формулою знаходимо десяткове значення кожної особини, а потім знаходимо значення функції і додаємо в масив значень.

Вихідний код створеної програми зображений в лістингу 2.1. Програма виконує 10 симуляцій виконання. Та виводить графік середніх значень популяцій. Графік зображено на рисунку 2.1.

Лістинг 2.1

import math

import random

import matplotlib.pyplot as plt

POPULATION\_SIZE = 34

ONE\_INDIVIDUAL\_SIZE = 34

MAX\_GENERATION = 20

pt\_crossover = 0.75

pm\_mutation = 0.001

fit\_value = []

BestNumber = 0

def getMiddleParents(values):

    sum = 0

    for value in values:

        sum += value

    sum = sum / POPULATION\_SIZE

    return sum

# fitness score calculation ............

def fitness\_score():

    global populations

    global fit\_value, medium\_values

    fit\_value = []

    for i in range(POPULATION\_SIZE):

        chromosome\_value = 0

        for j in range(ONE\_INDIVIDUAL\_SIZE - 1, -1, -1):

            chromosome\_value += populations[i][j] \* (2 \*\* (7 - j))

        print(f'chromosome\_value in 10 system = {chromosome\_value}')

        #func\_result = - (math.pow((chromosome\_value - 1), 2) / 256)

        func\_result = - \

            (math.pow((3\*math.pow(chromosome\_value, 2) - 12), 2) / 256)

        print(f'func res = {func\_result}')

        fit\_value.append(func\_result)

    # sorting to grow

    fit\_value, populations = zip(

        \*sorted(zip(fit\_value, populations), reverse=True))

    populations = list(populations)

    fit\_value = list(fit\_value)

    # find a medium value of population

    sum\_fit = getMiddleParents(fit\_value)

    medium\_values.append(sum\_fit)

    BestNumber = fit\_value[0]

    print(f'fit\_value {fit\_value}')

    print(f'population {populations}')

    print(f'best value {BestNumber}')

def selectparent():

    global parents, fit\_value, populations

    parents = []

    total\_sum = sum(fit\_value)

    print(f'sum {total\_sum}')

    if total\_sum == 0:

        return 0

    # find normalized form and create a list

    normalized\_fitness\_form = [i / total\_sum for i in fit\_value]

    # find cumulative fitness values for roulette wheel selection

    cumulative\_fitness\_value = []

    start = 0

    for norm\_value in normalized\_fitness\_form:

        start = start + norm\_value

        cumulative\_fitness\_value.append(start)

    print(f'normalized {normalized\_fitness\_form}')

    print(f'cumulative {cumulative\_fitness\_value}')

    # Формуємо нову популяцію батьків для схрещування, розмір рівний розміру популяції

    # Батьки можуть повторятися

    for count in range(POPULATION\_SIZE):

        # Крутимо рулетку, отримуємо число

        rand\_numb = random.uniform(0, 1)

        individual\_number = 0  # Номер батька

        # Перевіряємо співпадання числа з батьком

        for number\_parent, score in enumerate(cumulative\_fitness\_value):

            if rand\_numb <= score:

                # Додаємо батька у сформований список

                parents.append(populations[-(number\_parent + 1)])

                break

        individual\_number += 1

    print(f'parents len {len(parents)}')

    # for i in range(POPULATION\_SIZE):

    #     chromosome\_value = 0

    #     for j in range(ONE\_INDIVIDUAL\_SIZE - 1, -1, -1):

    #         chromosome\_value += parents[i][j] \* (2 \*\* (7 - j))

    #     print(f'chromosome\_value in 10 system = {chromosome\_value}')

def crossover():

    global parents

    #

    pairs\_count = int(len(parents) / 2)

    for pair in range(pairs\_count):

        chanse\_cross = random.randint(0, 100)

        if (chanse\_cross <= 61):

            # Вибираємо точку для одноточкового обміну

            cross\_point = random.randint(0, ONE\_INDIVIDUAL\_SIZE - 1)

            rand\_1 = random.randint(0, 7)  # Перший батько

            rand\_2 = random.randint(0, 7)  # Другий батько

            while rand\_1 == rand\_2:  #

                rand\_2 = random.randint(0, 7)

            # Формуємо частини для обміну

            fst\_parent\_bit = parents[rand\_1][cross\_point:]

            sec\_parent\_bit = parents[rand\_2][cross\_point:]

            print(f'par 1 {parents[rand\_1]} par2 {parents[rand\_2]}')

            # Створюємо нових батьків

            new\_parent1 = parents[rand\_1][:cross\_point] + sec\_parent\_bit

            new\_parent2 = parents[rand\_2][:cross\_point] + fst\_parent\_bit

            # Видаляємо старих батьків, ті що не розмножилися залишаються в списку

            if (rand\_2 > rand\_1):

                parents.remove(parents[rand\_2])

                parents.remove(parents[rand\_1])

            else:

                parents.remove(parents[rand\_1])

                parents.remove(parents[rand\_2])

            # Додаємо нових

            parents.append(new\_parent1)

            parents.append(new\_parent2)

            print(f'updated parents {parents}')

def mutation():

    global populations, parents

    mute = random.randint(0, 100)

    if mute <= 10:

        x = random.randint(0, POPULATION\_SIZE - 1)

        y = random.randint(0, ONE\_INDIVIDUAL\_SIZE - 1)

        if parents[x][y] == 1:

            parents[x][y] = 0

            print(f'Muted bit {parents[x][y]} in parent{parents[x]}')

        if parents[x][y] == 0:

            parents[x][y] = 1

            print(f'Muted bit {parents[x][y]} in parent{parents[x]}')

    populations = parents

if \_\_name\_\_ == "\_\_main\_\_":

    # initialize population

    populations = [[random.randint(0, 1) for x in range(

        ONE\_INDIVIDUAL\_SIZE)] for i in range(POPULATION\_SIZE)]

    parents = []

    medium\_values = []

    for i in range(MAX\_GENERATION):

        fitness\_score()

        if (selectparent()) == 1:

            break

        crossover()

        mutation()

    print(medium\_values)

    print(f'Best number is {BestNumber}')

    plt.plot(medium\_values, color="red")

    plt.xlabel('Покоління')

    plt.ylabel('Середня пристосованість')

    plt.title('Середня пристосованість в залежності від покоління')

    plt.show()

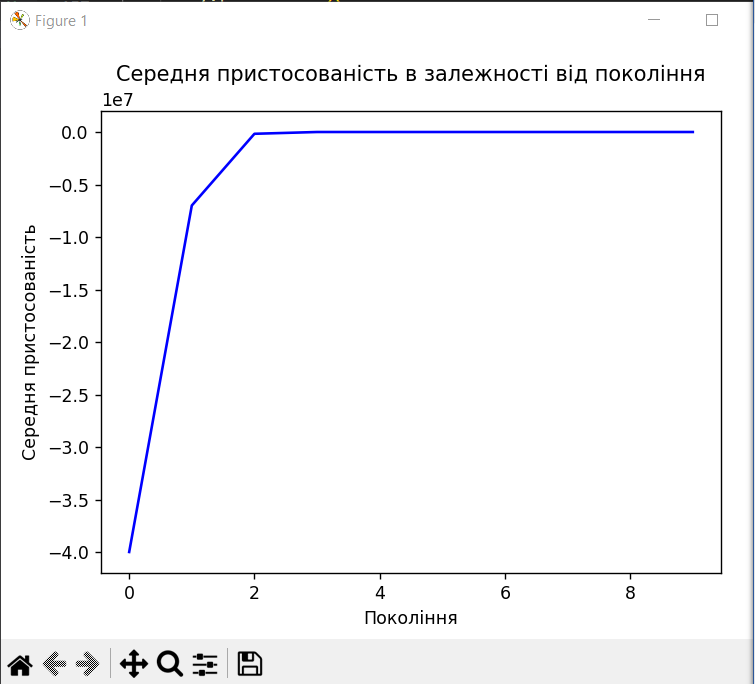


Рисунок 2.1 – Графік зміни популяцій в залежності від покоління

**Висновок**: на даній лабораторній роботі я навчився розв’язувати та виконувати пошук максимальних значень функції за допомогою генетичного алгоритму.